How to Write Correct Programs and Know It

H. D. MILLS

Chief Programmer Team Operations

IBM

COMPUTATION CENTER

FSC 73-5008

HOW TO WRITE CORRECT PROGRAMS AND KNOW IT

Abstract

There is no foolproof way to ever know that you have found the last error in a program. So the best way to acquire the confidence that a program has no errors is never to find the first one, no matter how much it is tested and used. It is an old myth that programming must be an error-prone, cut-and-try process of frustration and anxiety. But there is a new reality that you can learn to consistently write programs which are error free in their debugging and subsequent use. This new reality is founded in the ideas of structured programming and program correctness, which not only provide a systematic approach to programming but also motivate a high degree of concentration and precision in the coding subprocess.

February 1973

Federal Systems Division
INTERNATIONAL BUSINESS MACHINES CORPORATION
Gaithersburg, Maryland

CONTENTS

AN OLD MYTH AND A NEW REALITY

This article intends to explain literally how you can write correct programs and know that you have done so. It is to explode the myth that programming must be an error prone, cut and try process of frustration and anxiety which it has grown up to be in its short history.

By practicing the principles in this article you should be able to write correct programs and convince yourself and others that they are correct. Your programs should compile and execute properly the first time you try them, and from then on. Errors in either syntax or logic should be extremely rare, because you can prevent them from entering into your programs by positive actions on your part. Programs do not acquire bugs as people do germs-just by hanging around other buggy programs. They acquire bugs only by having programmers insert them.

There is a simple reason that you should expect your programs to be completely free of errors from the very start, for your own peace of mind. It is that you will never be able to prove that any one of your programs has no errors in it in a foolproof way. This is not because programs are so complex that it isn't worth the effort; it is because there simply is no way-logical or mathematical-to prove it, no matter how much effort you might put into it.

The ultimate faith you can have in one of your programs is your faith in the thought process that created it. With every error you find in testing and use, that faith is undermined. Even if you have found the last error left in your program, you cannot prove it is the last, and you cannot know it is the last.

So your real opportunity to know you have written a correct program is to never find the first error in it, no matter how much it is inspected, tested, and used.

Now the new reality is that ordinary programmers, with ordinary care, can learn to consistently write programs which are error-free from their inception. Just knowing that it is possible is half the battle. Learning how to write such programs is the other half. And gaining experience in writing such programs, small ones are first, then growing into larger ones, provides a new psychological basis for sustained concentration in programming, that is difficult to describe without direct personal experience.

CORRECTNESS AND CAPABILITY

Writing correct programs does not mean you can write programs once and for all. It means you can write programs to do exactly as you intend them to do. But as intentions change and you wish to add to or alter the capability of programs, program changes are required as well. The same opportunities and principles apply to these program changes. You should be able to modify programs correctly as well as write them correctly to begin with.

This distinction between correctness and capability is critical in understanding the new reality. Determining what a program should do is usually a much deeper problem than writing a program to do a predetermined process. It is the latter task that we now know to do correctly. For example, you

could wish to program a world champion chess player–that is a matter of capability, and a problem you may or may not be able to solve. Or you could wish to program a chess player whose move has been determined for every situation that can arise. You can write such a program correctly, but whether or not it becomes a world champion is another matter.

Our reason for distinguishing between correctness and capability is to identify what we can do and what we can't. As we grow in knowledge and experience in using a program, we frequently discover possibilities for improvement, so that the ability to correctly modify programs is as important as the ability to write them correctly.

THE DIFFICULTY WITH CORRECTNESS PROOFS

We start with a fundamental difficulty, which may seem fatal to our objective, but which paradoxically tells us what to do. There is no foolproof way to prove that a program is correct. There are, indeed, articles about ''the proof of correctness of programs,'' but such articles provide no foolproof proof, because no such proof can exist. The phrase ''proof of correctness'' is a relative term, meaning that a logical argument has been carried out with enough formality to justify the use of that phrase by today's editorial standards (which may vary from journal to journal, editor to editor, and will certainly vary in the future).

This fundamental difficulty is not in programming, but in mathematics–because the schoolboy idea of mathematics (or logic) as a body of supernatural verities and infallible procedures is simply not so. Mathematics is a human activity subject to human fallibilities. It has no basic secrets of truth or reason not known to chemists or sociologists. One simple example is in what we call the ''natural numbers,'' which are not natural at all. Everyone learns to count in the ''natural numbers'' from someone else, who learned to count from someone else, etc. But reaching

back far enough nobody knew how to count! The natural numbers are conscious human inventions, just as radios, Hamlet, and airplanes. They have survived because they work. And so it is with what the schoolboy learns of fractions, quadratic equations, calculus, etc., as though they were ''the truth, the whole truth, and nothing but the truth,'' when nothing could be farther from the truth.

The fact is that mathematics and its proofs consist of a body of human-devised, human-accepted definitions and procedures which finally rest on a mutual body of faith used in arriving at common beliefs about mathematical propositions. But no matter how formal or how rigorous we try to make these procedures, they still necessarily rest on foundations of pure faith. It is no inaccuracy to call mathematics ''the game of 'would you believe'.''

Even so mathematics is very useful, and we believe it to be largely correct in most of its development. It is correct enough to conduct business, design computers that run, and send men to the moon and back. And that is pretty good. It just isn't foolproof. Indeed, you should use all the mathematics you can to help convince yourself your programs are correct–to use today's phrase, ''to prove correctness'' as far as practicable. But you should do so knowing the limitations of mathematics yourself, and not looking for some schoolboy magic to replace your own responsibility.

ACQUIRING CONFIDENCE IN PROGRAMS

As we have seen, the ultimate confidence in a program is subjective, whether we realize it or not. If we believe a program is correct because of a formal proof of its correctness, the subjective confidence is in the proof methodology, and the further belief that this methodology applies to the full scope of the program. More often, our subjective confidence

is based on a combination of experiences in inspection (including formal proofs of correctness), testing, and usage.

If programming is regarded as a cut-and-try activity, a certain number of errors are expected in syntax and logic, and the testing and debugging phase is further expected to uncover most of these errors. But even as a cut-and-try activity, if the number of errors found in testing and debugging is excessive, the programmer becomes uneasy. Instead of being grateful for finding so many errors, he begins to doubt the thought processes that produced them. Many programmers recommend starting all over when this occurs.

On the other hand, as it happens occasionally, if a program, even in a cut and try activity, is free from error in all its testing and usage-with no debugging required-the subjective confidence of the programmer is remarkably affected. It will never be possible to prove such a program has no errors. But each hurdle it passes in testing and usage improves the plausibility that this is so, and that the thought processes that produced the program are holding up.

Thus, when you think about it, the real objective in programming should be to write correct programs from the start-not merely to merge from debugging with no errors. The new reality is that writing such correct programs from the start is a very possible human activity. And so it is that the very impossibility of foolproof proofs of the correctness of programs tells us what we must do. If no error ever occurs in a program, then a proof of correctness can tell us no more.

THE DISEASE OF SYNTAX ERRORS

With the advent of compilers and other debugging aids, it has been easy to adopt an attitude of ''let the compiler do it'' in finding errors of syntax. But in the long run, this is a devastating attitude because it fosters ignorance and carelessness that slides over to program logic that the compiler cannot uncover.

If your programming is a vocation rather than an avocation, there is no reason for you to take errors of syntax lightly in writing a program. Syntax errors are either errors of ignorance or carelessness. If they are errors of ignorance you need to do more homework on the syntax of your programming languages. If they are errors of carelessness you need to learn how to concentrate and take what you are doing more seriously.

A professional writer of English, or even a well-educated nonprofessional, has little trouble in writing complete sentences or remembering to end sentences with a period. Writing with syntatic precision is a simple necessity and practically an unconscious skill for any competent programmer. True enough the compiler will find syntax errors. But there are many times when a syntax error will be reinterpreted as a correct syntax for another statement so that a logic error results that neither the programmer or the compiler is aware of.

Writing correct syntax is like playing a perfect game of tic-tac-toe—not like sawing a board exactly in half, which requires perfect precision. It

is a combinatorial process which requires only a fixed and humanly possible degree of precision for correctness. For example, a complicated expression may end with five (or six) parentheses; but it will never end with 5.37521 ... parentheses. The difference between five and six is distinguishable in writing and reading, and whether it should be five or six depends only on previous characters of discrete kinds and locations in the expression.

The problem of writing correct program logic is more difficult than that of writing correct syntax and most of this article is about that problem. The reason for beginning with syntax errors is to identify an attitude of precision which will carry over with good effect into the problem of program logic. In fact, this emphasis on syntax is based on the reverse experience that, when programmers begin to get program logic correct from the start, the attitude of precision carries back into the coding, and they begin to get program syntax right from the start, with no special urging.

FUNCTIONS AS EXPRESSIONS OF ESSENTIAL PROGRAM LOGIC

A program operates on data, some of it intentional, and some of it often as a by-product of doing something else. For example, a program may operate on an array to recalculate its elements, but at the same time will calculate subscripts in order to identify specific elements of the array during execution. In particular, the last values for subscripts will be left laying around in memory. Ordinarly one will be interested in the array elements, and whether they are correctly computed, and not in the last values various subscripts happened to have. But in some cases computed data not central to the intention of the program may find a use in another program if its condition is known.

This picture of programs which operate on data, whether of central interest or not, arises naturally from viewing data as it occurs in machine storage. It is well known that such usage of data is one of the principal pitfalls in making larger and larger systems of programs work. It is the question of side effects, where some data not immediately visible at a program interface is altered or used.

The idea of a mathematical function allows one to be precise about the intentional effect of programs on data. For example, in the array case, its elements can be mapped into new elements using a functional description. Nothing is said about subscripts or even if subscripts are used in the computation of its elements. In this form, anyone else is forewarned that any assumptions about subscripts is done at his own peril and is probably untrue.

The usual first encounter with the idea of a function is one which relates two variables, say; a function f which relates y to x in the form

$$y = f(x)$$

where:

$$f(x) = x^2 + 3x + 2.$$

For our purposes in dealing with finite but complex combinatorial objects the more modern definition of a function as a set of ordered pairs is more useful. For example, we may write $(x,y) \in f$ instead of $y = f(x)$ to emphasize the set aspect. Since a function is set, the ordinary set operations apply to functions. The expression $x^2 + 3x + 2$ is a rule which defines which ordered pairs belong to f; in fact, it is only one of many possible rules. In programming, functional specification corresponds to function, and program corresponds to rule.

For the example above, we can use set notation to describe f as

$$f = \{(x,y) \mid y = x^2 + 3x + 2\}.$$

The variables x, y are dummy variables, since:

$$g = \{(u, v) \mid v = u^2 + 3u + 2\}$$

or:

$$h = \{(u, v) \mid v = (u + 1)(u + 2)\}$$

are both identical to f. In this context we can see that the three rules correspond to three different programs (different operations and different variables) that realize the same functional specification.

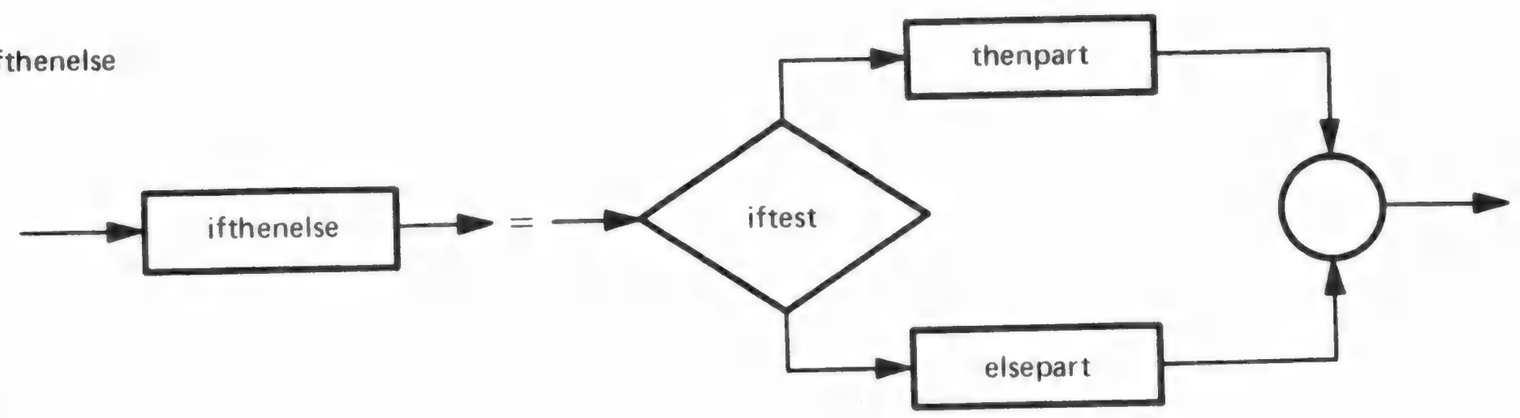
STRUCTURED PROGRAMMING AND PROGRAM CORRECTNESS

You can write programs with correct function logic by using principles of structured programming and program correctness which are applied in your line-by-line program construction. A programmer begins with a functional specification which describes what the program to be is to do. In his mind he converts that specification into program statements and then verifies that the statements created in fact do what the program was intended to do. In structured programming there is a precise description of this mental activity. It begins with the functional specification and repeatedly dissects it, a step at a time, into new functional subspecifications connected by program statements until the program is complete. It does not consist of a large leap in faith from a functional specification to loose collection of program statements which are fitted piece-by-piece into a program. The structured programming process analyzes functional specifications rather than synthesizing program statements. One brief way of understanding structured programming and how to prove the correctness of programs written in this way is this:

a. Any functional specification can be defined in terms of a mathematical function which maps inputs into outputs without regard to its internal construction. We show such a function (functional specification) as

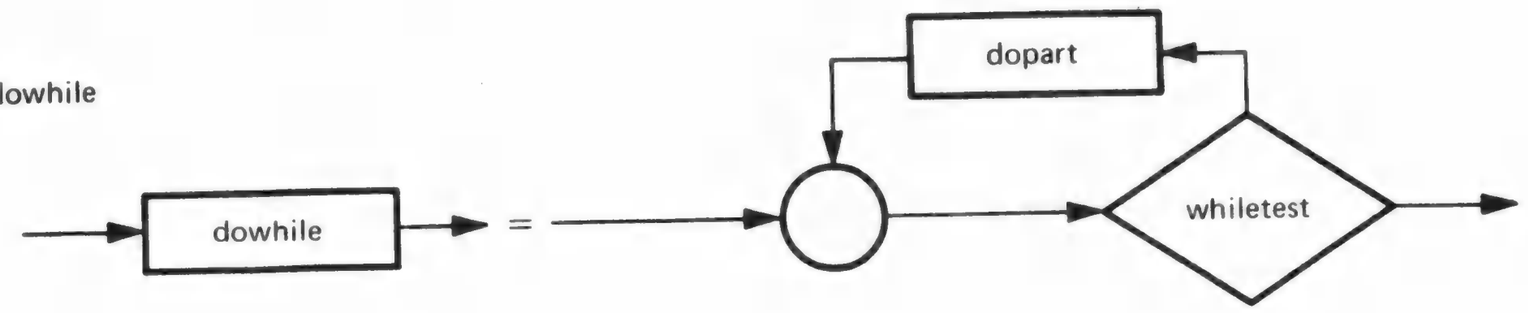
b. Any flowchartable program used to realize a function is equivalent to a structured program which can be constructed by the repeated use of only these three basic program figures:

1. ifthenelse



2. sequence

3. dowhile



Each ifthenelse, sequence, and dowhile on the left side is a function realized in a new way on the right side. Each thenpart, . . ., dopart on the right side is just a new function and can be replaced by another ifthenelse, sequence, or dowhile figure in a subsequent expansion step.

The structured program construction process proceeds from an original functional specification as a series of decisions, which specify which figure and what resulting new tests and functions are required to expand the original and any intermediate functions required. When the functions required can be written directly as program statements, the expansion process is completed.

In a language such as PL/I, these expressions can be written directly in matching statements, without labels or GOTO's to result finally in a GOTO-free program. Such a GOTO-free program can be read sequentially, without jumping around. The relationship between program text and execution thus becomes expecially clear.

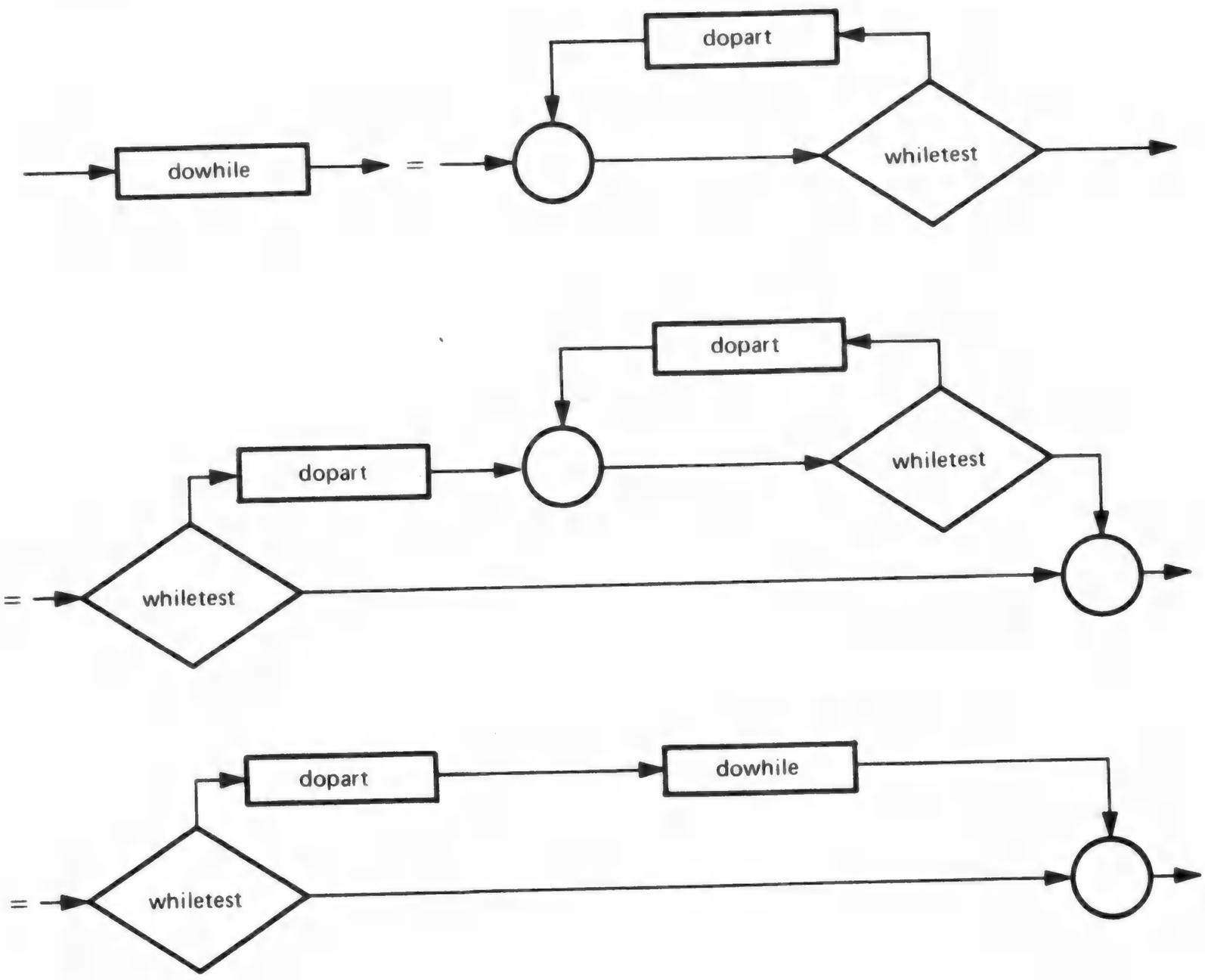
c. At each expansion step, the correctness of that step can be decided by answering a standard question that goes with that type of expansion. If the answer is yes, the step is correct and the program expansion can proceed. If the answer is no, the step is not correct and a new one should be defined right then. We will see that the questions are:

1. ifthenelse—Whenever the iftest is true, does the thenpart do the ifthenelse; and whenever the iftest is false, does the elsepart do the ifthenelse?

2. sequence-Does the firstpart followed by the secondpart do the sequence?

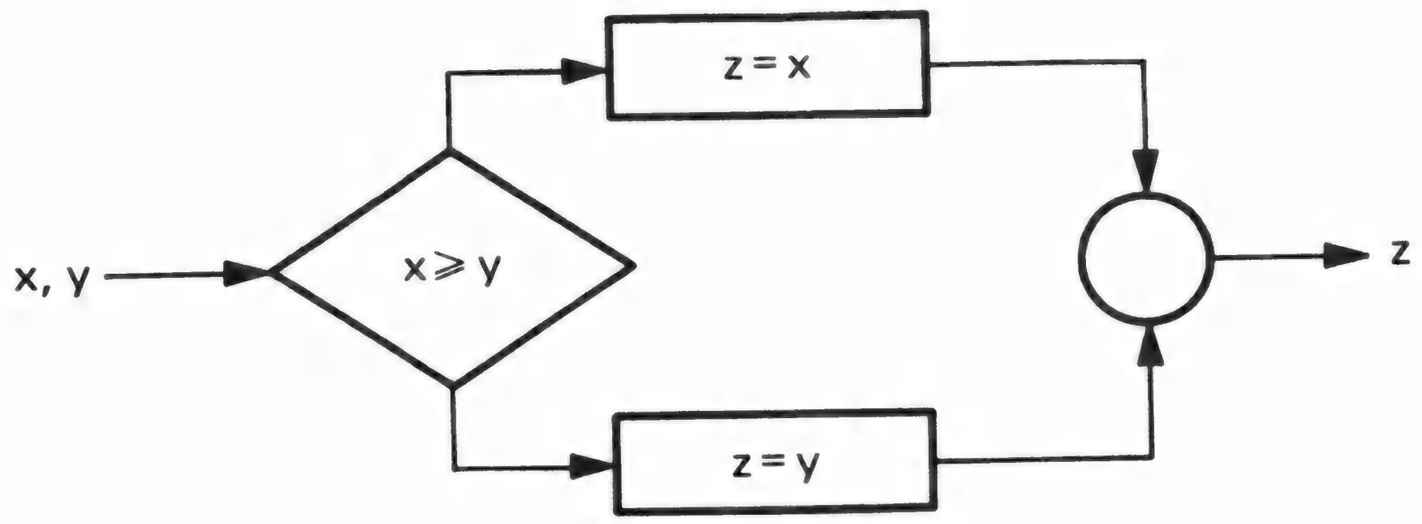
3. dowhile-Whenever the whiletest is true, does the dopart followed by the dowhile do the dowhile; and whenever the whiletest is false, does the identity function (no-op program) do the dowhile?

The questions for the ifthenelse and sequence expansions are self-evident. The question for the dowhile becomes self-evident by observing this sequence of equivalent expansions: Expand the execution of the dowhile into an ifthenelse, and then observe that the dowhile reappears as the secondpart of the sequence making up the thenpart; the elsepart is the identity.



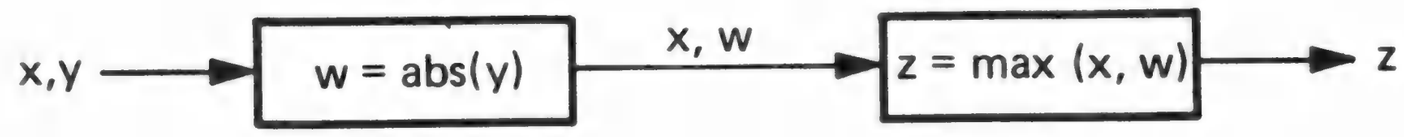
d. When steps 2 and 3 are carried out to the point where no sub-specifications remain, the result is a complete program and the proof of its correctness has been completed as well. Some illustrations of individual steps with their correctness questions are:

1. ifthenelse: z = max (x, y)



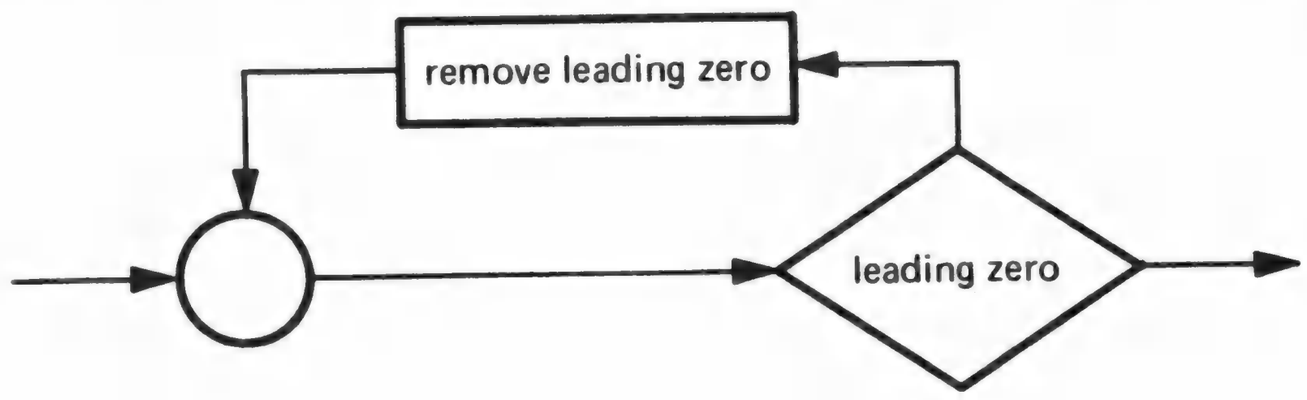
whenever x ≥ y does z = x do z = max (x, y) and
whenever x < y does z = y do z = max (x, y)?

2. sequence: z = max (x, abs(y))



does w = abs(y) followed by z = max (x, w)
do z = max(x, abs (y))?

3. dowhile: remove leading zeros (from a decimal integer)



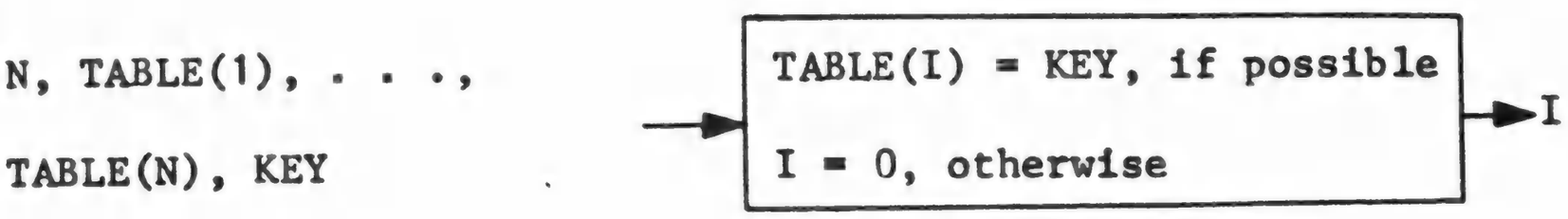
Whenever there is a leading zero, does leading zero followed by remove leading zeros do remove leading zeros; and whenever there is no leading zero, does doing nothing do remove leading zeros?

WRITING SOME SEARCHES

In order to see these principles in action, consider the problem of searching for an item called KEY in a list called TABLE, with a total of N elements, denoted TABLE (1), . . ., TABLE (N), respectively; we are to display the results of the search in an item called I, which is to satisfy the relation

TABLE (I) = KEY, if possible
I = 0, otherwise.

Note we have defined a function in words. The argument is N + 2 items namely N, TABLE (1), . . ., TABLE (N), KEY, and the value is I, as diagrammed



It is easy to invent a program, say in PL/I, for this function.

```
SEARCH1: PROCEDURE;
   I = 0;
   DO J = 1 TO N;
      IF TABLE(J) = KEY THEN
         I = J;
   END;
END SEARCH1:
```

It is not an efficient program, to be sure, but it seems to be correct. Why? First, it is a sequence of two subprograms whose functions are

a. firstpart: set I to zero

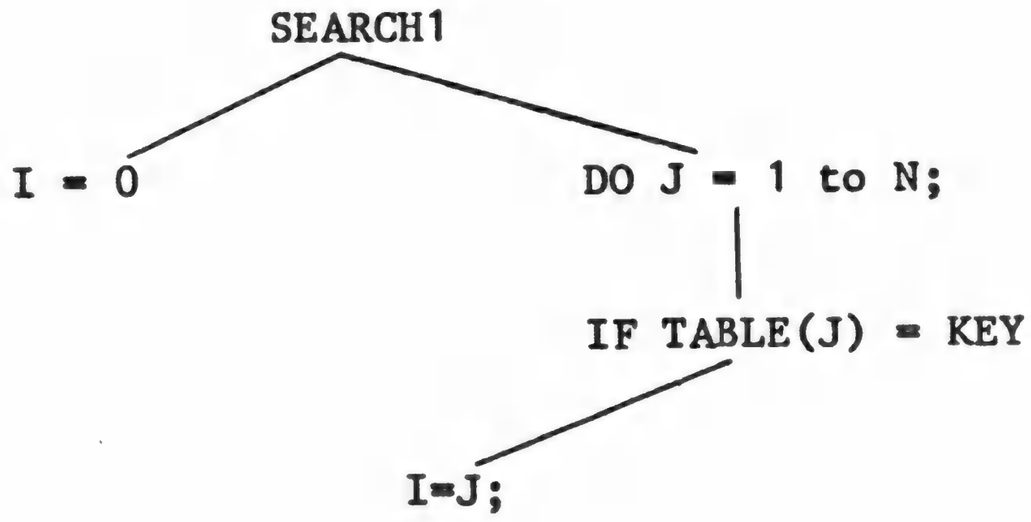
b. secondpart: find, if possible, a value for I for which
TABLE (I) = KEY;
otherwise, leave I unchanged

The sequence question, above, asks if firstpart followed by secondpart does the sequence. It is believed so. Next, the secondpart above is itself a loop, but not a dowhile figure. Instead it is the familiar indexed loop, which we will call doloop for short. It is worth our attention as an extra convenience beyond the three basic figures given above, under an extra point of discipline. This extra discipline is that the index of the doloop is not altered in any way by the dopart of the doloop. Then the doloop becomes an extended sequence, with a firstpart, secondpart, . . ., nthpart. The corresponding correctness question is a simple extension of the sequence question as well. The dopart in this case is

dopart: if TABLE(J) = KEY then
set I to J, otherwise leave I unchanged

and it is easy to see that the sequence of such doparts, for J = 1, ..., N indeed does the doloop (secondpart above). Finally, the dopart, is itself an ifthen (ifthenelse with null else) figure, and it is easy to see that it satisfies its functional requirement.

In summary, we have articulated an analysis a programmer does at a glance to illustrate the building blocks of a skilled observer. In this case they are structured in a tree of the form



and each node defines a subfunction and subprogram simultaneously. A skilled pianist has learned to play scales and arpeggios with little attention to the individual notes, and a skilled programmer also learns to put small combinations of statements together in almost the same way. But even so, the basic questions are valid and need consideration explicitly. If the answers are obvious, they will not take much time to verify; if not obvious, they are worth looking into.

It is easy to see how to improve SEARCH 1 to SEARCH 2 as follows:

```
SEARCH2:  PROCEDURE;
     I = 0;
     DO J = 1 to N WHILE (I = 0);
          IF TABLE(J) = KEY THEN
               I = J;
          END;
     END SEARCH2;
```

whereas SEARCH1 looked at every item in TABLE, whether successful or not part way through, SEARCH2 has enough sense to stop looking at the first success in TABLE. The only change in SEARCH2 is the WHILE clause. The effect is a doloop with a conditional termination, which can be rewritten as:

```
J = 1
DO WHILE (J <= N & I = 0);
     IF TABLE(J) = KEY THEN
          I = J;
     J = J + 1;
END;
```

That is, the doloopwhile becomes a sequence of a firstpart for initialization and secondpart of dowhile whose dopart includes incrementing the index. In this form the dowhile question applies–it asks:

> whenever $J \leq N$ and $I = 0$, does the dopart followed by the dowhile (with $J = J + 1$ now) do the dowhile; and whenever $J > N$ or $I \neq 0$, does doing nothing do the dowhile?

We can see that it does. If KEY has not yet been found in TABLE, and we have not looked at every item, then we can look at the next item and set I, J accordingly and still complete the task required of the dowhile.

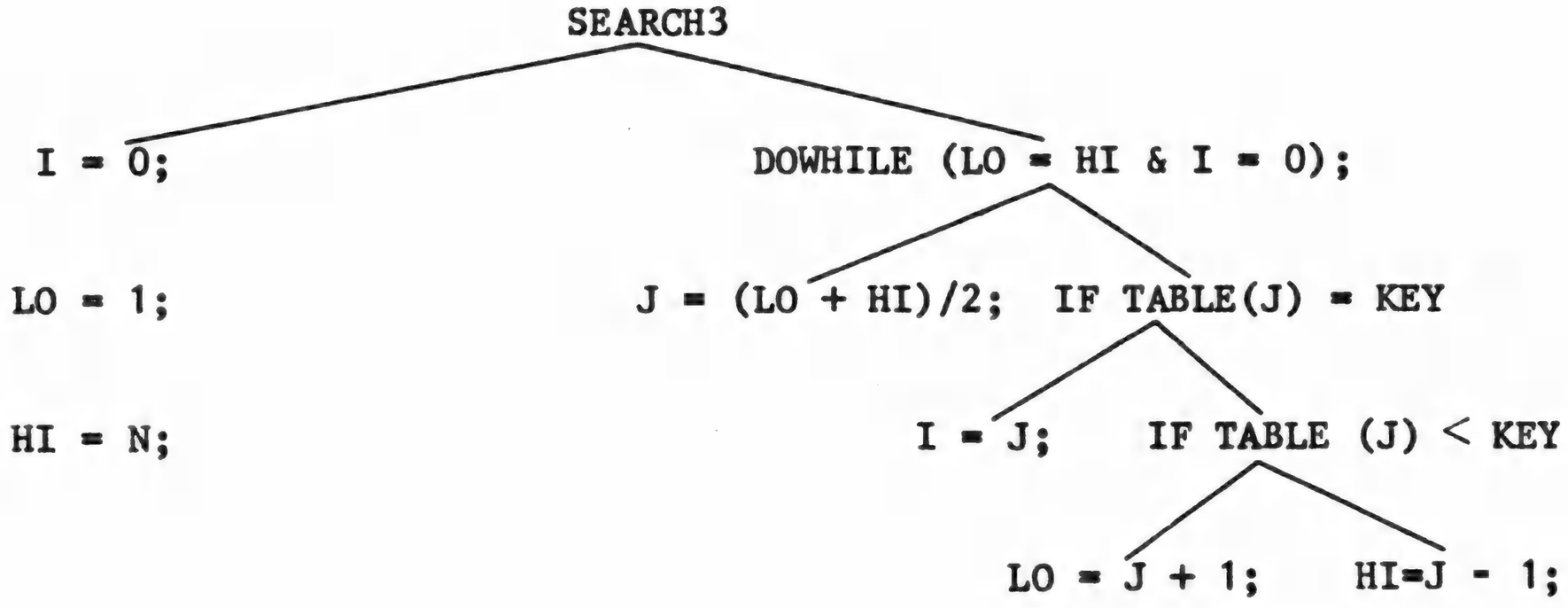
Looking back to the functional idea in programming discussed earlier, note also that the improved SEARCH2 leaves an unpredictable value for J, whereas SEARCH 1 left $J = N + 1$ always. If a programmer took the program rather than the functional specification, as definitive, he could be in trouble by depending on a value for J.

If the elements of TABLE are sorted (say, in ascending order), then a possibly more efficient search can be defined as a binary search. By examining an element as close as possible to the middle of the table, we either find KEY, or we then know that KEY is to be found only in one half or the other of the remaining table. That basic step can be repeated in the half indicated and continued until a table of only one element is reached. If KEY is not found by that step, it does not exist in the table.

We put the foregoing in a program, as follows, introducing variables LO and HI which define the lower and upper subscript of the table being searched at each step.

```
SEARCH3:  PROCEDURE;
     I = 0;
    LO = 1;
    HI = N;
    DOWHILE (LO <= HI & I = 0);
          J = (LO + HI)/2;
          IF TABLE(J) = KEY THEN
               I = J;
          ELSE
               IF TABLE (J) < KEY THEN
                    LO = J + 1;
               ELSE
                    HI = J - 1;
        END;
END SEARCH3;
```

The tree of questions about SEARCH3 is given by:



There are five nonterminal nodes in this tree, corresponding to two sequences, one dowhile, and two ifthenelses. (Note, we regard a sequence of assignments as simply one generalized assignment for our purposes here.) Each node defines a function which in turn serves as a component in the next function.

The function for SEARCH3 is the same as that stated already, namely, given N, TABLE(1), . . ., TABLE (N), KEY find I which satisfies the relation

TABLE(I) = KEY, if possible,
I = 0, otherwise.

The function for the dowhile is, given N, TABLE(1), . . ., TABLE (N), KEY, LO $\geq$ 1, HI $\leq$ N, find I which satisfies:

TABLE(I) = KEY and LO $\leq$ I $\leq$ HI, if possible,
I unchanged otherwise.

It is clear, with the three initialization assignments, that this sequence (of initialization and the dowhile) does SEARCH3.

Next, the dopart of the dowhile is a sequence. The function of this dopart is, given N, TABLE(1), . . ., TABLE(N), KEY, LO, HI, find I, LO, HI so that:

I = (LO + HI)/2 and TABLE(I) = KEY, if possible
or I is unchanged and
LO is changed to (LO + HI)/2 + 1
if TABLE ((LO + HI) /2) < KEY
and HI is changed to (LO + HI)/2 - 1
if TABLE ((LO + HI)/2) > KEY

In order to see that the dowhile is accomplished by this whiletest and this dopart, two principal considerations are needed. First, doing the dopart once without finding KEY cannot prevent the dowhile finding KEY, if it is possible; second, doing the dopart sufficiently many times (finitely) guarantees the ultimate failure of the whiletest. For the first consideration, the assumption of a sorted TABLE must be invoked, with the verification that each failure to find KEY in the TABLE ensures that KEY will not be found above, or below, that point as the case may be. For the second consideration, it is sufficient to consider the algebraic difference, HI - LO, and to observe that when I remains 0 (otherwise the whiletest is false), then either LO or HI is changed so that:

$$(HI - LO)_{after} < (HI - LO)_{before} - 1$$

so that eventually,

$$HI - LO < 0$$

and the whiletest will fail.

We won't elaborate the arguments for the two ifthenelses. But note their functions are defined by the preceding arguments for the dowhile. Note also, the essential difference in the argument for a dowhile and for an indexed doloop.

IN STRUCTURED PROGRAMMING

The first name in structured programming is Edsger W. Dijkstra, who early recognized the problem of dealing with complexity in the programming process, and identified the practical means of maintaining control logic discipline through the three basic figures defined above. In the note ''GOTO statement considered harmful'' [1] Dijkstra pointed out the problem of relating program text and execution, and how the yet-to-be-named structured programming discipline helped to solve that problem. In the paper ''A Constructive Approach to Program Correctness'' [2], Dijkstra illustrated the stepwise development of programs by expanding functional specifications into text. In the paper ''Structured Programming'' [3], Dijkstra further elaborated the stepwise development idea, and pointed out the reduction in mental verification effort required in structured programs in comparison with general unstructured programs.

The first published proof that structured programs were sufficiently powerful to represent any flowchartable program logic was developed by Giuseppe Jacopini in a paper ''Flow Diagrams, Turning Machines, and Languages with Only Two Formation Rules'' [4], coauthored with Corrado Böhm. As the title suggests, the main point of the paper lies outside programming, and the paper is difficult to read. But a construction is outlined there for converting general flowcharts into structured ones.

The result of Jacopini is given a simpler, more explicit development by Harlan D. Mills in the report ''Mathematical Foundations for Structured Programming'' [5] and called the ''structure theorem'' there. In

the paper ''Top-Down Programming In Large Systems'' [6] Mills identified a stepwise development process for large programming systems.

IN PROGRAM CORRECTNESS

The program correctness questions defined above first appeared in a so-called ''correctness theorem'' in the ''Mathematical Foundations for Structured Programming'' of Mills. Earlier, independent, equivalent approaches to program correctness were developed by Peter Naur in ''Proof of Algorithms by General Snapshots'' [7], and by Robert Floyd in ''Assigning Meanings to Programs'' [8].

The work of Naur and Floyd applies to general unstructured programs and involves the identification of all possible cycles in a flowchart, with a cut point in each cycle, and the invention of a so-called ''inductive predicate'' for each such cut point, which must be shown to remain valid for one tour of each such cycle. With cycles the question of termination also arises to be handled separately.

The correctness theorem of Mills refers only to structured programs (although the Jacopini construction permits its eventual application to general programs), and exploits the resulting simplicity of the basic figures. As pointed out by Dijkstra [3], the number of cycles in a general program grows exponentially in size, while the number of correctness questions for structured programs grows only linearly.

Program correctness can be expected to be an important area in future software development. Ralph London has given proofs of correctness for programs of significant size and complexity in ''Certification of Algorithm 245 Treesort 3: Proof of Algorithms-A New Kind of Certification'' [9], and in ''Correctness of a Compiler for a LISP Subset'' [10]. Although our emphasis is informal above, if you are serious about programming you should understand how to prove correctness more formally

when the occasion warrants, for example, using London's ideas in ''Proving Programs Correct: Some Techniques and Examples'' [11].

IN PROGRAMMING PRACTICES

In addition to Dijkstra's paper ''A Constructive Approach to Program Correctness,'' already mentioned, Niklaus Wirth's paper ''Program Development by Stepwise Refinement'' [12] illustrates the approach to the so-called ''eight queens problem.'' C.A.R. Hoare proves the correctness of an almost structured program in ''Proof of a Program: FIND'' [13], which is also illuminating.

A major application of structured programming (in conjunction with certain organizational techniques) was carried out and described by F.T. Baker in ''Chief Programmer Team Management of Production Programming'' [14] and in ''System Quality through Structured Programming'' [15]. Baker reports a substantial increase in productivity and an even more remarkable decrease in error incidence in the development of a large conversational information system.

The stepwise refinement approach to program development can be identified as a system design methodology as well, as discussed in an early paper by Fred Zurcher and Brian Randell, ''Iterative Multi-Level Modelling-A Methodology for Computer System Design'' [16]. In ''A Design Methodology for Reliable Software Systems'' [17] Barbara Liskov combines principles of structured programming and program correctness into a systematic approach to program development.

IN MATHEMATICS

Professor Wilder, in ''Evolution of Mathematical Concepts-An Elementary Study'' [18], points out (pp. 196f.) ''It appears to be a universal

phenomenon in the evolution of culture, that when a culture has evolved sufficiently to achieve a certain degree of maturity, there then arises a need among its participants for an 'explanation' of its origin. . . the mathematical subculture of modern western culture furnished no exception... the faith in the 'truth' of mathematical theories that has been sustained in the general culture is shared to a considerable extent by the mathematical subculture.'' Professor Wilder then goes on to conclude that mathematics will continue to evolve just as any other human activity—on the basis of its value to the human condition.

REFERENCES

1. Dijkstra, E.W., ''GOTO Statement Considered Harmful,'' Communications of the ACM, Vol. 11, No. 3, (March 1968) pp. 147-148.

2. Dijkstra, E.W., ''A Constructive Approach to the Problem of Program Correctness,'' BIT, Vol. 8, No. 3, (1968) pp. 174-186.

3. Dijkstra, E.W., ''Structured Programming,'' Software Engineering Techniques, NATO Science Committee (Eds. J.N. Burton, and B. Randell), (1969) pp. 88-94.

4. Böhm, Corrado, and Jacopini, Giuseppe, ''Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules,'' Communications of Association for Computing Machinery, Vol. 9, (1966) pp. 366-371.

5. Mills, H.D., ''Mathematical Foundations for Structured Programming,'' FSC 72-6012, (February 1972).

6. Mills, H.D., ''Top-Down Programming in Large Systems,'' Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, NYU (Ed. Randall Rustin) (1971).

7. Naur, P., ''Proof of Algorithms by General Snapshots,'' BIT Vol. 6, (1966) pp. 310-316.

8. Floyd, R.W., ''Assigning Meanings to Programs,'' Proceedings of the Symposium in Applied Mathematics, Vol. 19, (Ed. J.T. Schwartz), American Mathematical Society, Providence, R.I., (1967) pp. 19-32.

9. London, R.L., ''Certification of Algorithm 245 Treesort 3: Proof of Algorithms-A New Kind of Certification,'' Comm. ACM 13, (1970) pp. 371-373.

10. London, R.L., ''Correctness of a Compiler for a LISP Subset,'' Proceedings of the Association for Computing Machinery Conference on Proving Assertions About Programs, New Mexico State University, Las Cruces, N.M., (January 1972) pp. 51-57.

11. London, R.L., ''Proving Programs Correct: Some Techniques and Examples,'' BIT, Vol. 10, No. 2, (1970) pp. 168-182.

12. Wirth, Niklaus, ''Program Development by Stepwise Refinement,'' Communications of the Association for Computing Machinery 14, No. 4, (April 1971), pp. 221-227.

13. Hoare, C.A.R., ''Proof of a Program: FIND,'' Communications of the Association for Computing Machinery, Vol. 14, No. 1, (January 1971) pp. 39-45.

14. Baker, F.T., ''Chief Programmer Team Management of Production Programming,'' IBM Systems Journal, Vol. 11, No. 1, (1972) pp. 56-73.

15. Baker, F.T., ''System Quality Through Structured Programming,'' AFIPS Conference Proceedings, Vol. 41, Part I, (1972) pp. 339-343.

16. Zurcher, F., and Randell, B., ''Iterative Multi-Level Modelling-A Methodology for Computer System Design,'' Proceedings of the IFIP Congress, (1968), pp. D138-D142.

17. Liskov, Barbara H., ''A Design Methodology for Reliable Software Systems,'' Proceedings of Fall Joint Computer Conference,'' AFIPS, Vol. 41, Part 1, (1972) pp. 191-199.

18. Wilder, Raymond L., Evolution of Mathematical Concepts-An Elementary Study, John Wiley and Sons, Inc., New York, (1968).

COMPUTATION CENTER

IBM

Printed in U.S.A.